COMP311: *COMPUTER ORGANIZATION!*

Lecture 20: More Stack Review and RISC-V Calling Convention

tinyurl.com/comp311-sp26

Logistics Update!

- Optional WS 4 posted now
- WA6 will be posted today
- Lab 4 due tomorrow

RISC-V WARM UP

```
    addi t1, x0, 0      # sum = 0
    addi t2, x0, 0      # i = 0
    la   t3, arr        # t3 = &arr[0]

loop:
    slli t4, t2, 2      # i * 4
    add  t5, t3, t4     # address of arr[i]
    lw   t6, 0(t5)      # t6 = arr[i]

    beq  t6, x0, done   # stop when arr[i] == 0

    andi t7, t6, 1      # t7 = arr[i] & 1
    beq  t7, x0, next   # skip if even

    add  t1, t1, t6     # sum += arr[i]

next:
    addi t2, t2, 1      # i++
    j    loop

done:
    halt
```

initialization

computing idx & access elts

loop condition

if cond

accumulation of sum

1101
& 0001
0001

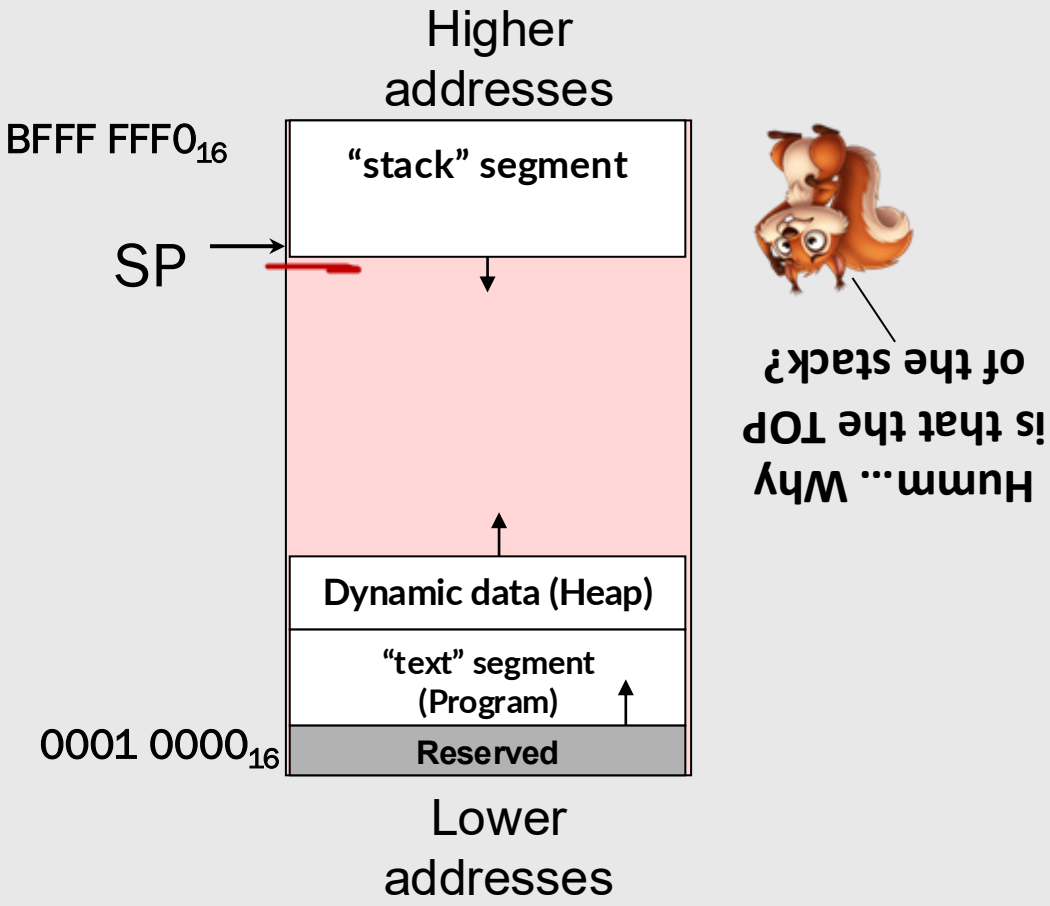
STACK CONCEPTS AND CALLING CONVENTION REVIEW ☺

RISC-V Stack Convention

These conventions for allocating variables when calling fns → part of the ABI

CONVENTIONS:

- **Assign a register as the Stack Pointer (sp = x2).**
- **Stack grows DOWN (towards lower addresses) on *pushes* and *allocates***
- ***sp* points to the last or TOP *used* location.**
- **Stack is placed far away from the program and its data.**



Register use conventions

By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages.

We'll use these names increasingly.

Why have such conventions?

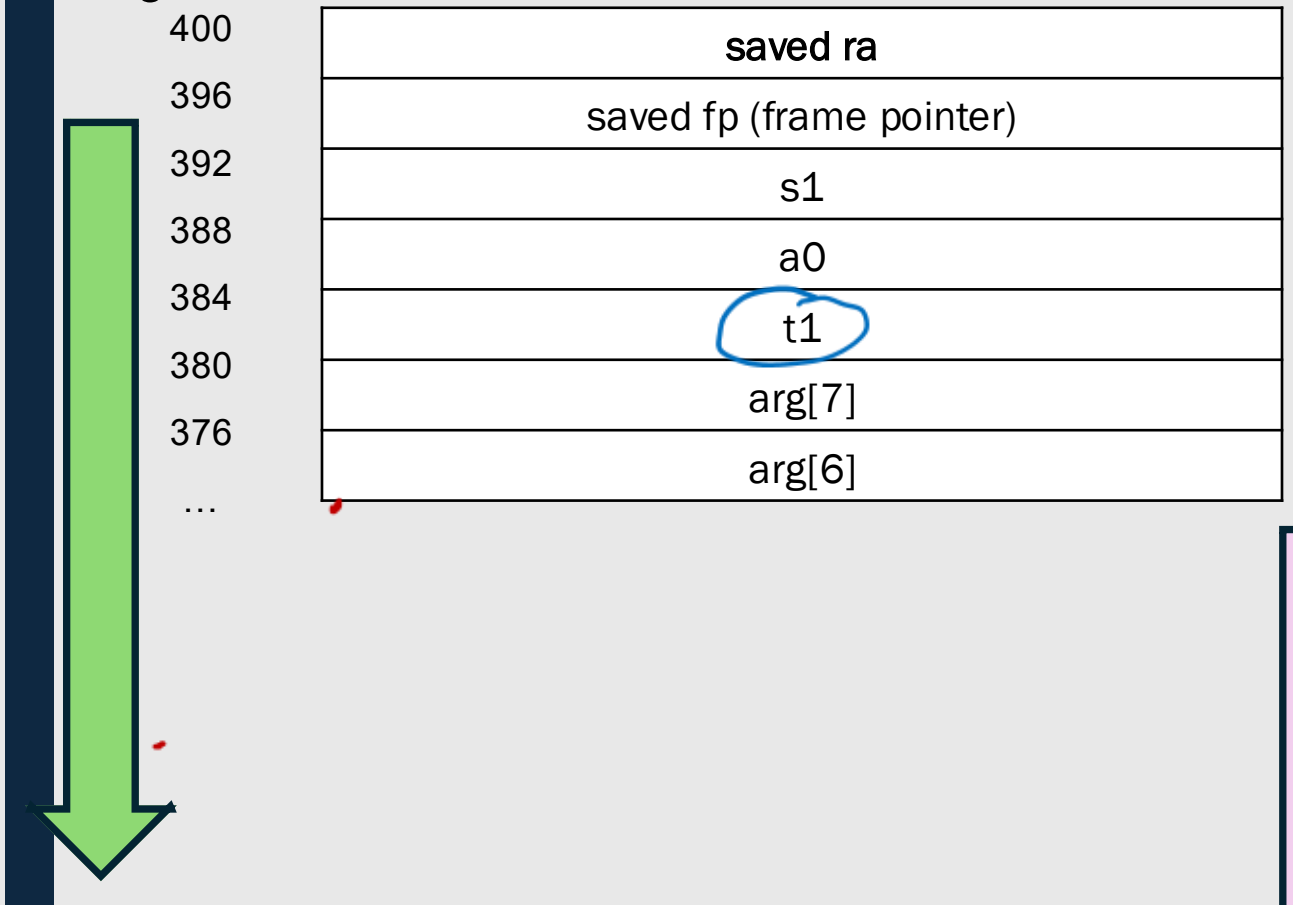
x0/zero (always zero)	x16/a6 (argument)
x1/ra (return address)	x17/a7 (argument)
x2/sp (stack pointer)	x18/s2 (saved)
x3/gp (global pointer)	x19/s3 (saved)
x4/tp (thread pointer)	x20/s4 (saved)
x5/t0 (temporary)	x21/s5 (saved)
x6/t1 (temporary)	x22 (saved)
x7/t2 (temporary)	x23 (saved)
x8/fp (frame pointer)	x24 (saved)
x9/s1 (saved)	x25 (saved)
x10/a0 (argument/return value 1)	x26 (saved)
x11/a1 (argument/return value 2)	x27 (saved)
x12/a2 (argument)	x28 (temporary)
x13/a3 (argument)	x29 (temporary)
x14/a4 (argument)	X30 (temporary)
x15/a5 (argument)	X31 (temporary)

Prevents functions from accidentally overwriting each other's values

Allows independently compiled code to co-execute correctly!

Separately written fns can safely share CPU without corrupting each other's data

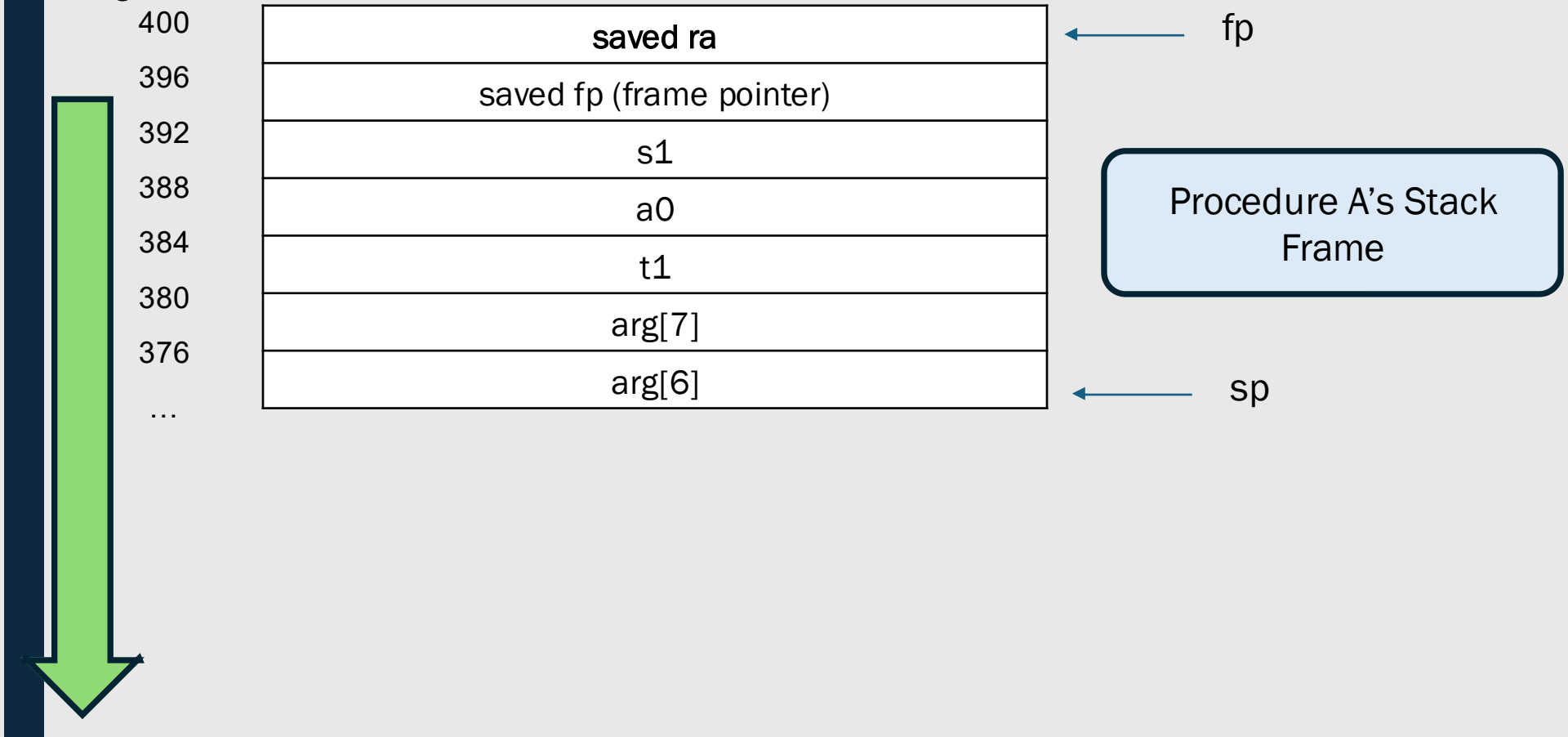
Higher addresses



- Stack grows down!
- **Stack pointer (register sp/x2): p**oints to the **current top (lowest address) of the stack** and changes as values are pushed or popped.
- **Frame pointer:** (fp/x8/s0) Provides a **stable** reference point within the **current stack frame** so data can be accessed with fixed offsets.

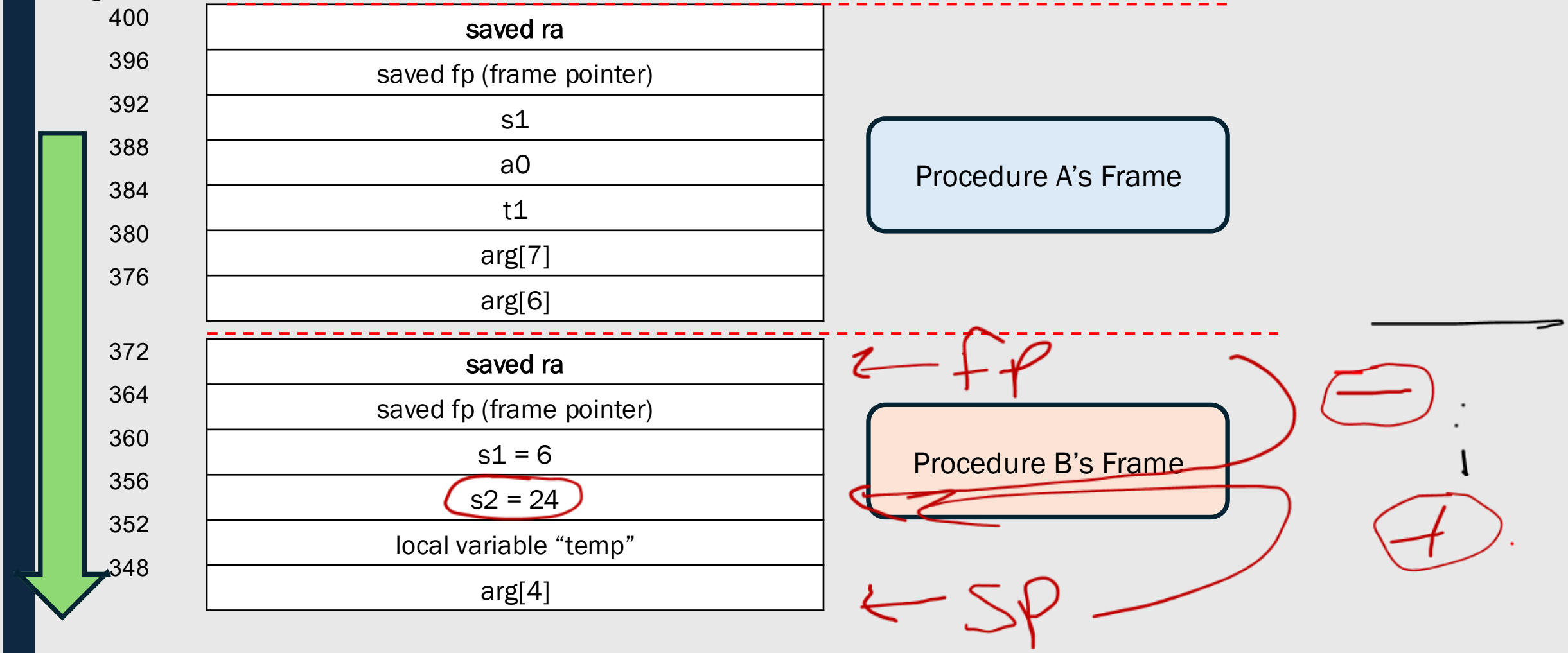
Lower addresses

Higher addresses



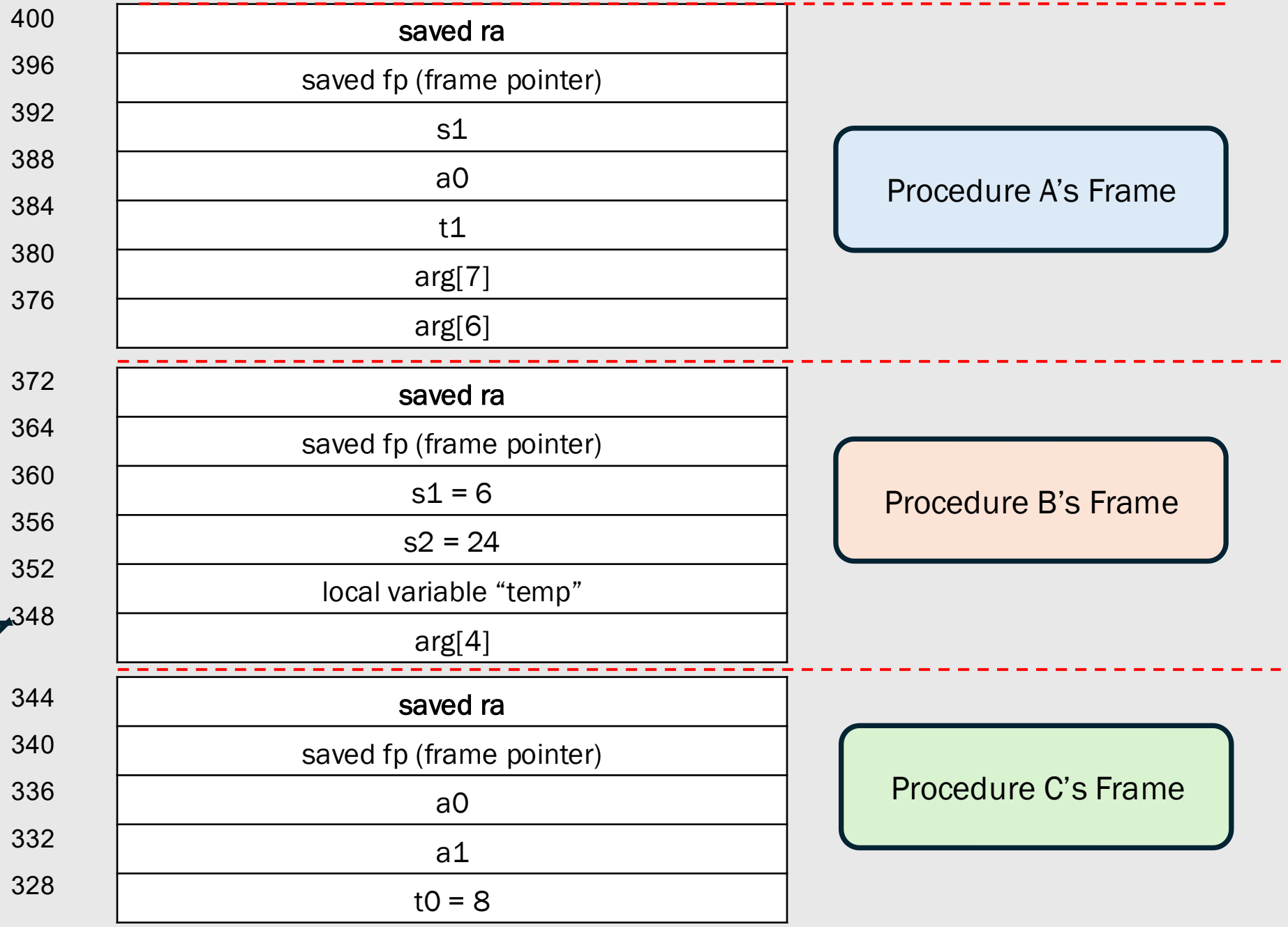
Lower addresses

Higher addresses



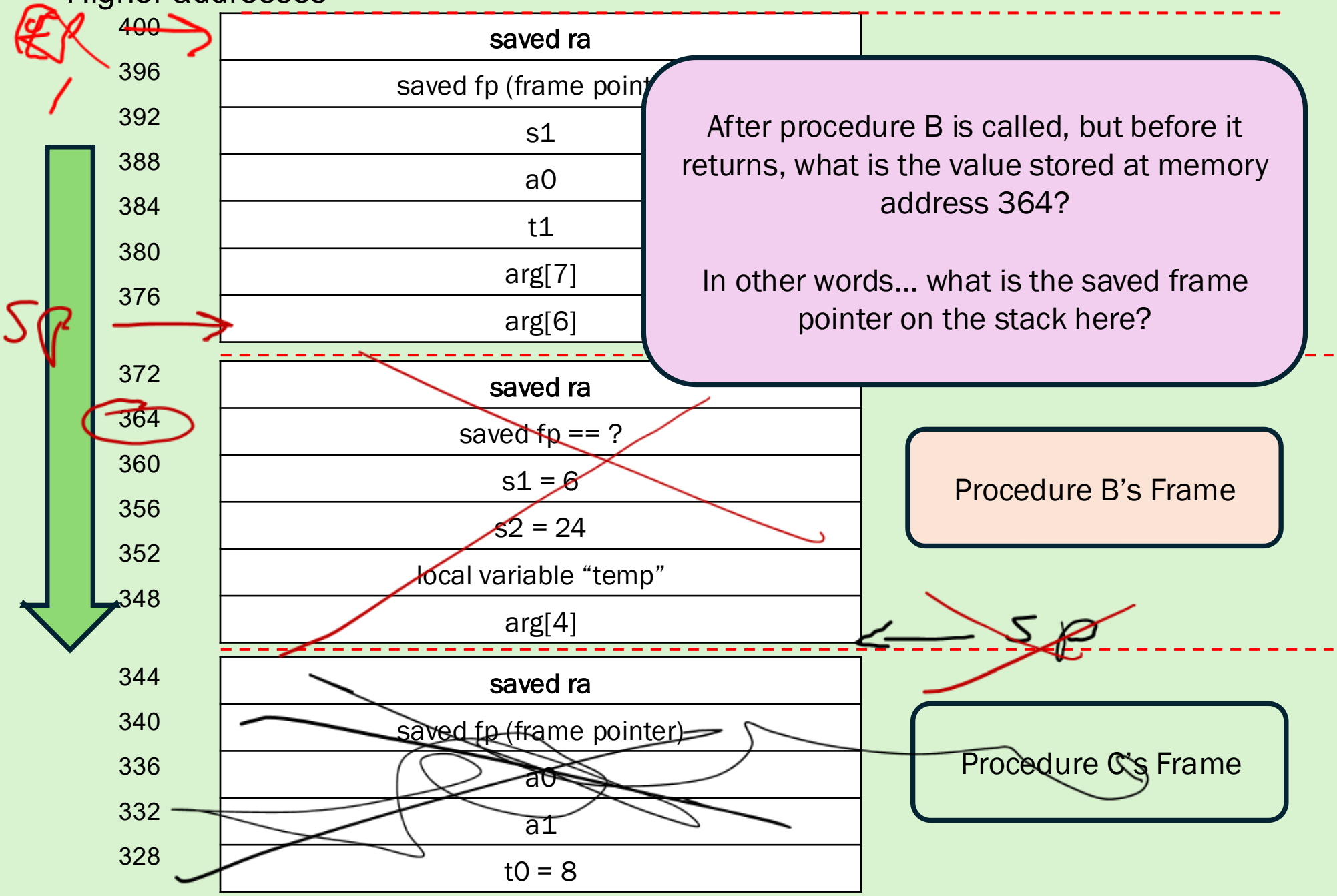
Lower addresses

Higher addresses



Lower addresses

Higher addresses



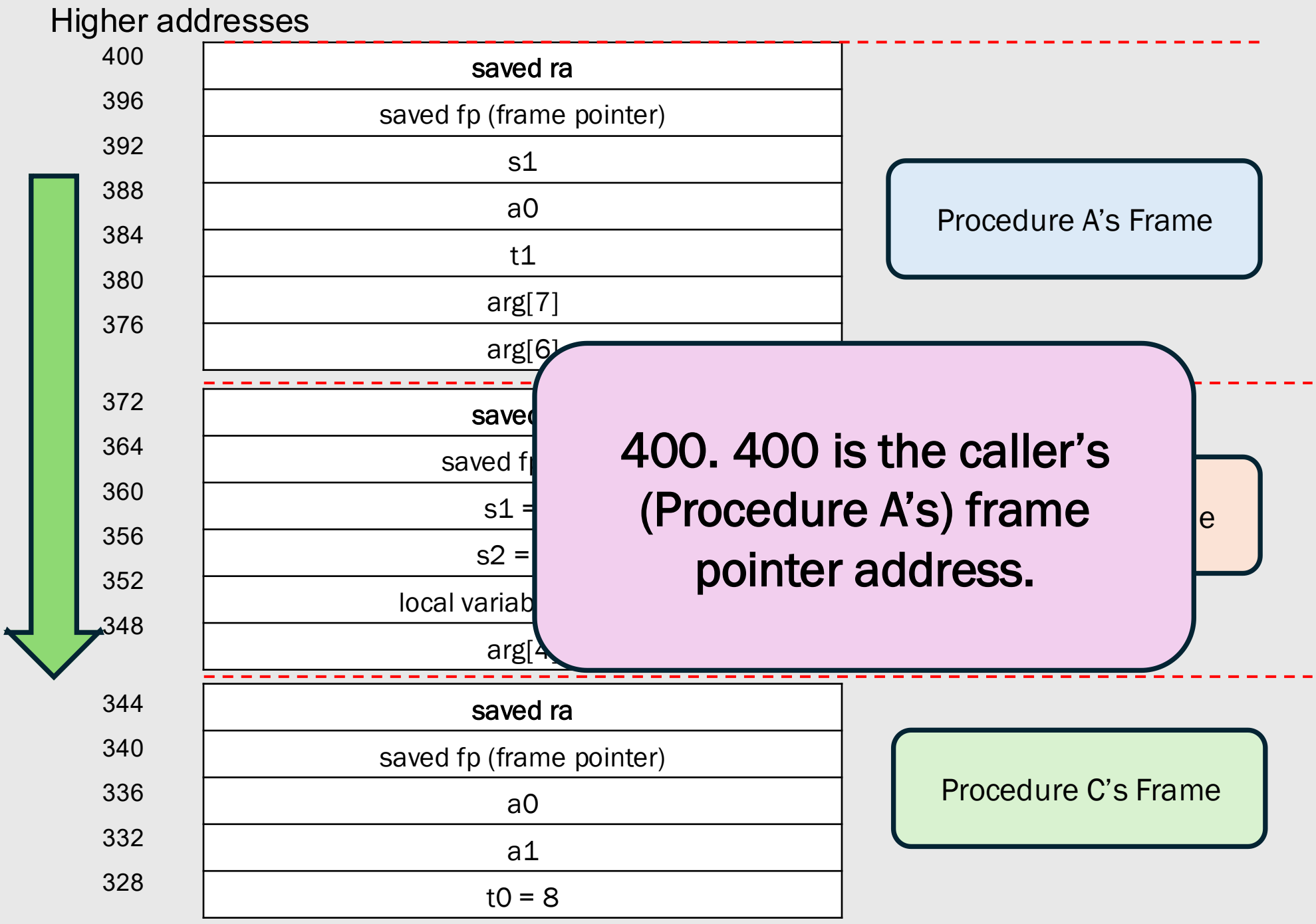
After procedure B is called, but before it returns, what is the value stored at memory address 364?

In other words… what is the saved frame pointer on the stack here?

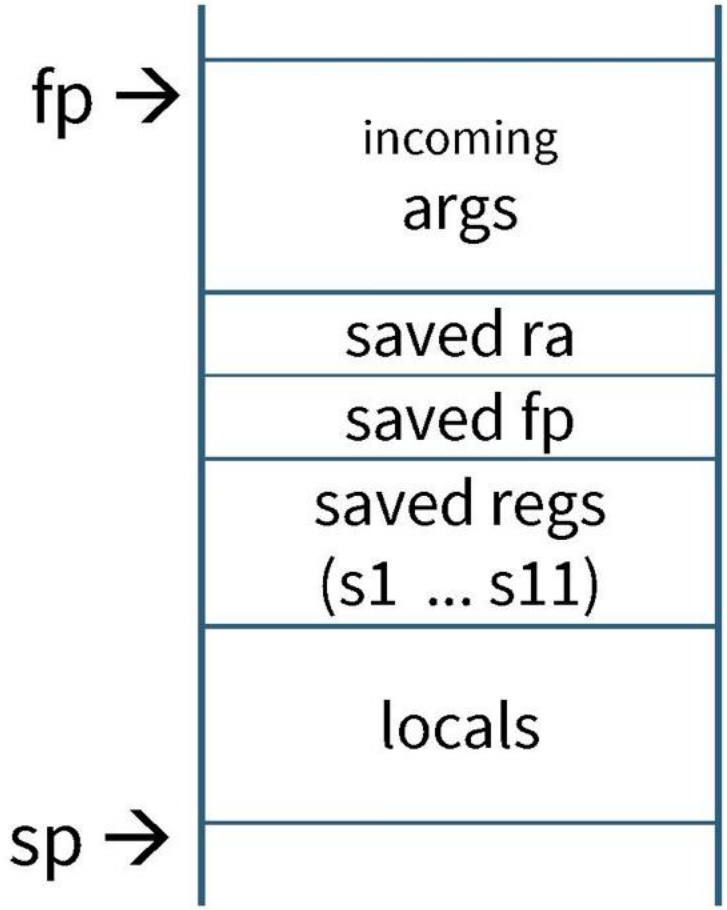
Procedure B’s Frame

Procedure C’s Frame

Lower addresses


Higher addresses
400
396
392
388
384
380
376
saved ra
saved fp (frame pointer)
s1
a0
t1
arg[7]
arg[6]
Procedure A's Frame
372
364
360
356
352
348
344
340
336
332
328
saved ra
saved fp (frame pointer)
a0
a1
t0 = 8
Procedure C's Frame
400. 400 is the caller's (Procedure A's) frame pointer address.
Lower addresses

RISC-V Calling Convention Cheat Sheat/Summary



Source:
https://www.cs.cornell.edu/courses/cs3410/2025fa/notes/asm-call.html

- *first eight* args passed in registers `a0`, `a1`, ..., `a7`
- Space for args passed in *childs's* stack frame
- return value (if any) in `a0`, `a1`
- stack frame at `sp`
  - contains `ra` (clobbered on JAL to sub-functions)
  - contains `fp`
  - contains *local vars* (possibly clobbered by sub-functions)
  - contains *space for incoming args*
- *Saved registers* (callee save regs) are *preserved*
- *Temporary registers* (caller save) regs are *not*

RISC-V Registers

- Return address: `x1` (`ra`)
- Stack pointer: `x2` (`sp`)
- Frame pointer: `x8` (`fp/s0`)
- First eight arguments: `x10-x17` (`a0-a7`)
- Return result: `x10-x11` (`a0-a1`)
- Callee-save free regs: `x18-x27` (`s2-s11`)
- Caller-save free regs: `x5-x7`, `x28-x31` (`t0-t6`)
- Global pointer: `x3` (`gp`)
- Thread pointer: `x4` (`tp`)

Stack Management Policies

sw rx, 4(sp)

ALLOCATE k: reserve k WORDS of stack

SP = SP - 4*k

```
addi    sp,sp,-4*k
```

DEALLOCATE k: release k WORDS of stack

SP = SP + 4*k

```
addi    sp,sp,4*k
```

PUSH x: push Reg[x] onto stack

Mem[SP - 4] = Rx

SP = SP - 4

```
addi    sp,sp,-4
sw      rx,(sp)
```

POP x: pop the top of the stack into Reg[x]

Rx = Mem[SP]

SP = SP + 4

```
lw      rx,(sp)
addi    sp,sp,4
```

Code interacts with the stack in 3 main ways:

- At the beginning of the function, it will "*push* a stack frame onto the call stack" by moving SP downward to make space for its own stack frame.

- During the execution of the function, it will use (positive) offsets from sp or fp to locate each of its local variables.

- At the end of the function, before it returns, it will "*pop* the stack frame off the call stack" by moving sp back up to wherever it used to be, "destroying" its stack frame.

Code interacts with the stack in 3 main ways:

- At the beginning of the function, it will "*push* a stack frame onto the call stack" by moving SP downward to make space for its own stack frame.

- During the execution of the function, it will use (positive) offsets from sp or fp to locate each of its local variables.

- At the end of the function, before it returns, it will "*pop* the stack frame off the call stack" by moving sp back up to wherever it used to be, "destroying" its stack frame.

Note: according to the ABI `fp` is just an alias for s0. So we use s0 here in case you want to try it in the assembler

```
proc1:
    addi sp, sp, -8        # allocate space for ra and s0
    sw   ra, 4(sp)         # save return address
    sw   s0, 0(sp)         # save old frame pointer
    addi fp, sp, 4         # set frame pointer

    addi sp, sp, ___-16___     # allocate space for s1–s4

    sw   s1, ___12___(sp)     # save s1
    sw   s2, ___8___(sp)      # save s2
    sw   s3, ___4__(sp)       # save s3
    sw   s4, ___0___(sp)      # save s4

    jal  ra, proc2         # call another procedure

    lw   s1, ___-8___(s0)     # restore s1
    lw   s2, ___-12___(s0)    # restore s2
    lw   s3, ___-16__ (s0)    # restore s3
    lw   s4, ___-20___(s0)    # restore s4

    addi sp, s0, 4         # restore sp
    lw   ra, 0(s0)         # restore ra
    lw   s0, -4(s0)        # restore old frame pointer
    jalr x0, 0(ra)         # return
```

Code interacts with the stack in 3 main ways:

- At the beginning of the function, it will "*push* a stack frame onto the call stack" by moving SP downward to make space for its own stack frame.
- During the execution of the function, it will use (positive) offsets from sp or fp to locate each of its local variables.
- At the end of the function, before it returns, it will "*pop* the stack frame off the call stack" by moving sp back up to wherever it used to be, "destroying" its stack frame.

```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi s0, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)      # save s2
   sw   s3, ___4__(sp)       # save s3
   sw   s4, ___0___(sp)      # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)    # restore s2
   lw   s3, ___-16__ (s0)    # restore s3
   lw   s4, ___-20___(s0)    # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Code interacts with the stack in 3 main ways:

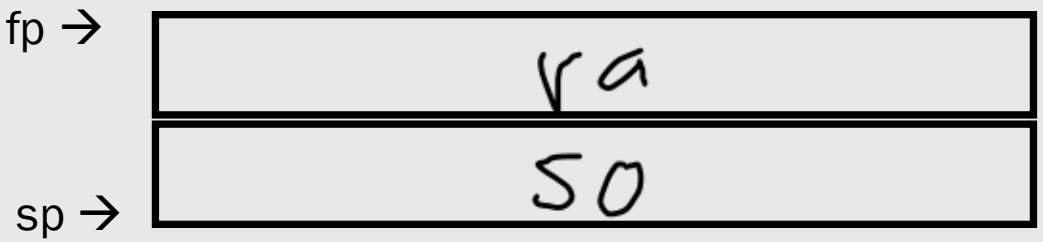
- At the beginning of the function, it will "*push* a stack frame onto the call stack" by moving SP downward to make space for its own stack frame.

- During the execution of the function, it will use (positive) offsets from sp or fp to locate each of its local variables.

- At the end of the function, before it returns, it will "*pop* the stack frame off the call stack" by moving sp back up to wherever it used to be, "destroying" its stack frame.

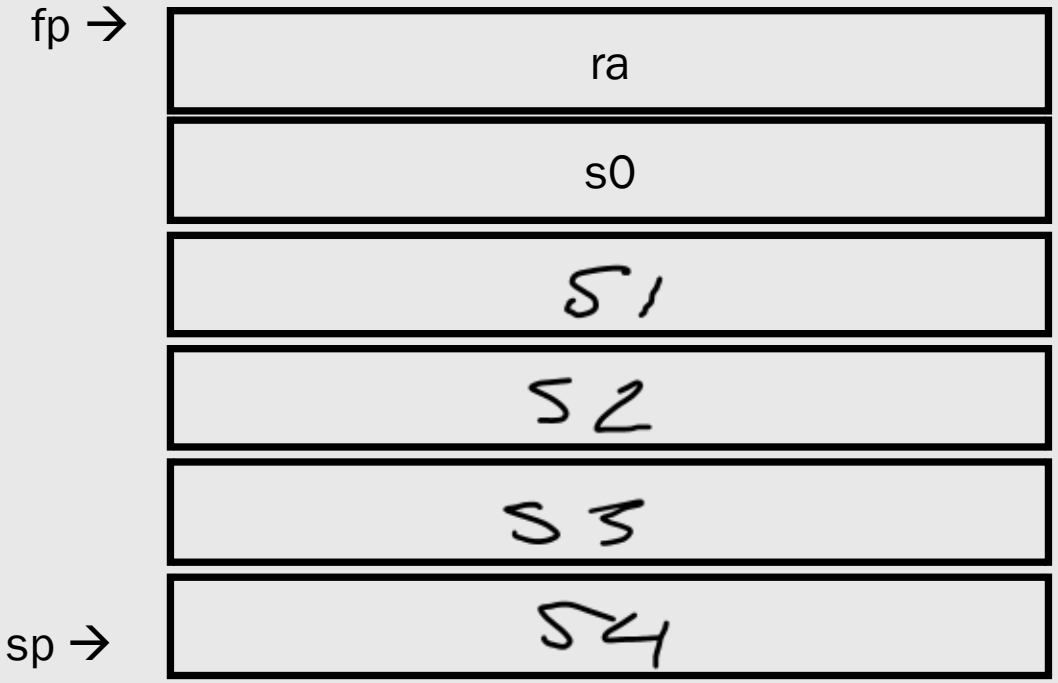
```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi fp, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)      # save s2
   sw   s3, ___4__(sp)       # save s3
   sw   s4, ___0___(sp)      # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)    # restore s2
   lw   s3, ___-16__ (s0)    # restore s3
   lw   s4, ___-20___(s0)    # restore s4

   addi sp, s0, 4         # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)        # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Review




```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi fp, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)     # save s2
   sw   s3, ___4__(sp)     # save s3
   sw   s4, ___0___(sp)     # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)     # restore s2
   lw   s3, ___-16__ (s0)     # restore s3
   lw   s4, ___-20___(s0)     # restore s4

   addi sp, s0, 4          # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)         # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Review

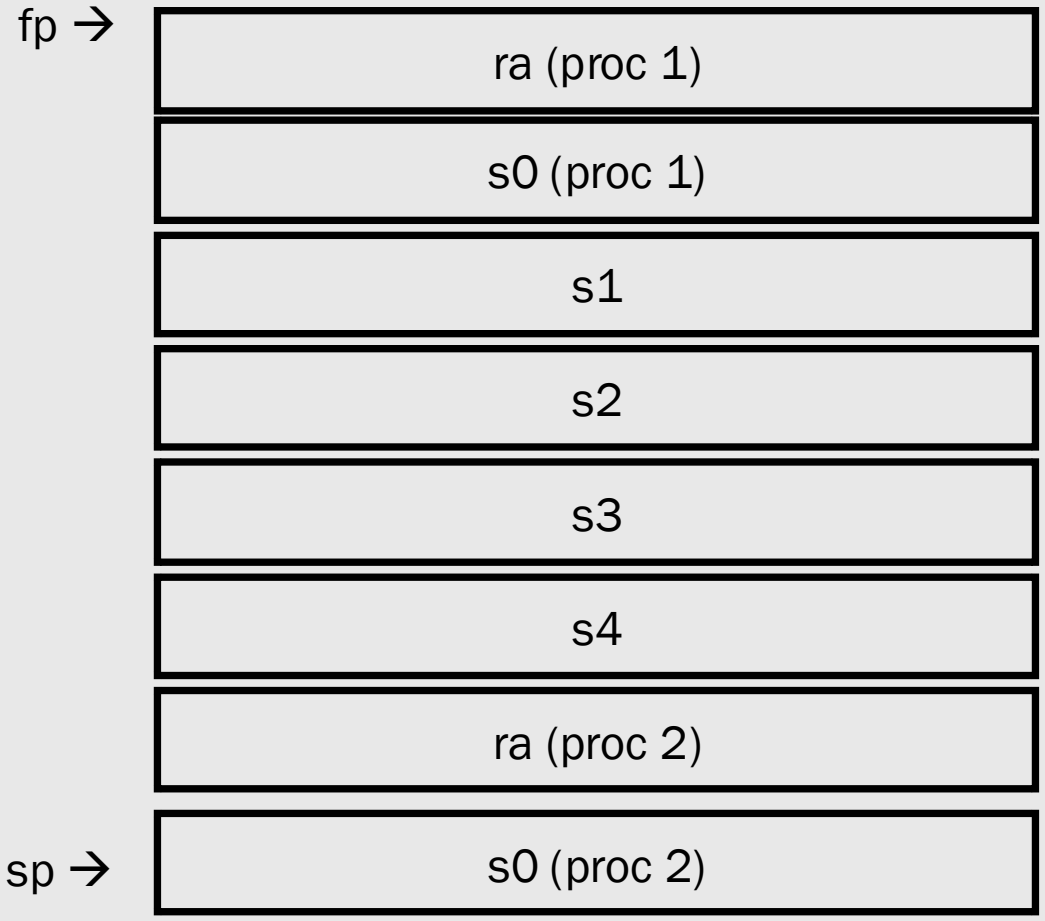



```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi fp, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)     # save s2
   sw   s3, ___4__(sp)     # save s3
   sw   s4, ___0___(sp)     # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)     # restore s2
   lw   s3, ___-16__ (s0)     # restore s3
   lw   s4, ___-20___(s0)     # restore s4

   addi sp, s0, 4          # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)         # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Review

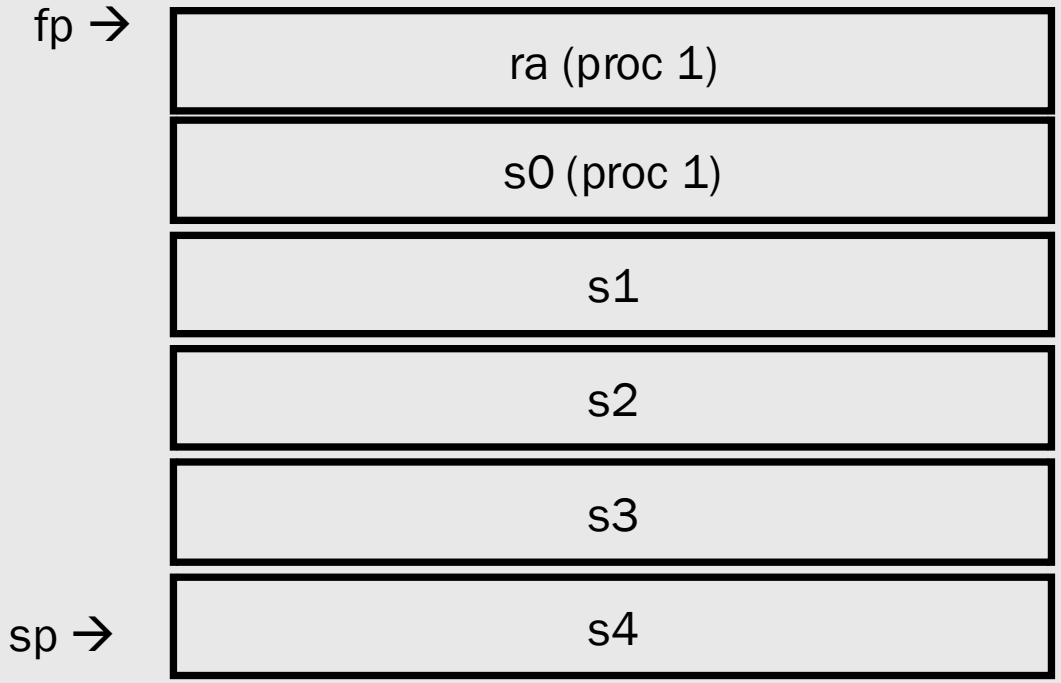

```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi fp, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)     # save s2
   sw   s3, ___4__(sp)     # save s3
   sw   s4, ___0___(sp)     # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)     # restore s2
   lw   s3, ___-16__ (s0)     # restore s3
   lw   s4, ___-20___(s0)     # restore s4

   addi sp, s0, 4          # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)         # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Review




```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi fp, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)     # save s2
   sw   s3, ___4__(sp)     # save s3
   sw   s4, ___0___(sp)     # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)     # restore s2
   lw   s3, ___-16__ (s0)     # restore s3
   lw   s4, ___-20___(s0)     # restore s4

   addi sp, s0, 4          # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)         # restore old frame pointer
   jalr x0, 0(ra)         # return
```

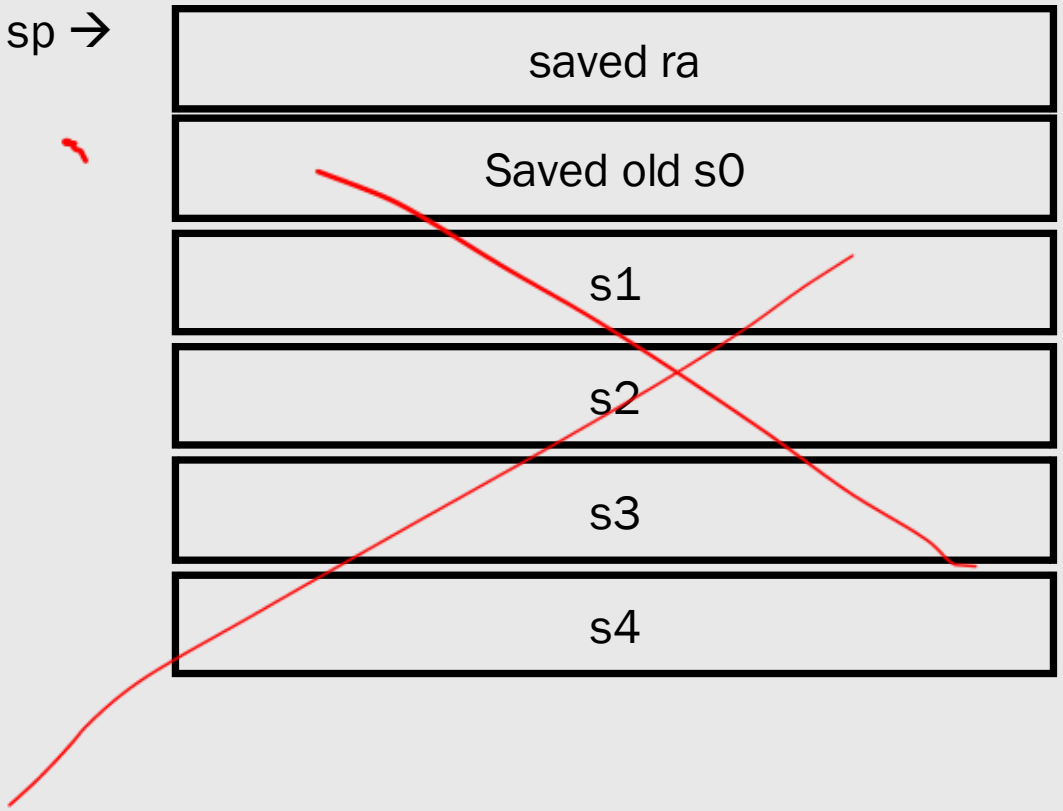
Review



s0 holds the address of our frame pointer

```
proc1:
   addi sp, sp, -8        # allocate space for ra and s0
   sw   ra, 4(sp)         # save return address
   sw   s0, 0(sp)         # save old frame pointer
   addi fp, sp, 4         # set frame pointer

   addi sp, sp, ___-16___     # allocate space for s1–s4

   sw   s1, ___12___(sp)     # save s1
   sw   s2, ___8___(sp)     # save s2
   sw   s3, ___4__(sp)     # save s3
   sw   s4, ___0___(sp)     # save s4

   jal  ra, proc2         # call another procedure

   lw   s1, ___-8___(s0)     # restore s1
   lw   s2, ___-12___(s0)     # restore s2
   lw   s3, ___-16__ (s0)     # restore s3
   lw   s4, ___-20___(s0)     # restore s4

   addi sp, s0, 4          # restore sp
   lw   ra, 0(s0)         # restore ra
   lw   s0, -4(s0)         # restore old frame pointer
   jalr x0, 0(ra)         # return
```

Review




```
proc1:
    addi sp, sp, -8        # allocate space for ra and s0
    sw   ra, 4(sp)         # save return address
    sw   s0, 0(sp)         # save old frame pointer
    addi fp, sp, 4         # set frame pointer

    addi sp, sp, ___-16___     # allocate space for s1–s4

    sw   s1, ___12___(sp)     # save s1
    sw   s2, ___8___(sp)     # save s2
    sw   s3, ___4__(sp)     # save s3
    sw   s4, ___0___(sp)     # save s4

    jal  ra, proc2         # call another procedure

    lw   s1, ___-8___(s0)     # restore s1
    lw   s2, ___-12___(s0)     # restore s2
    lw   s3, ___-16__ (s0)     # restore s3
    lw   s4, ___-20___(s0)     # restore s4

    addi sp, s0, 4          # restore sp
    lw   ra, 0(s0)         # restore ra
    lw   s0, -4(s0)         # restore old frame pointer
    jalr x0, 0(ra)         # return
```

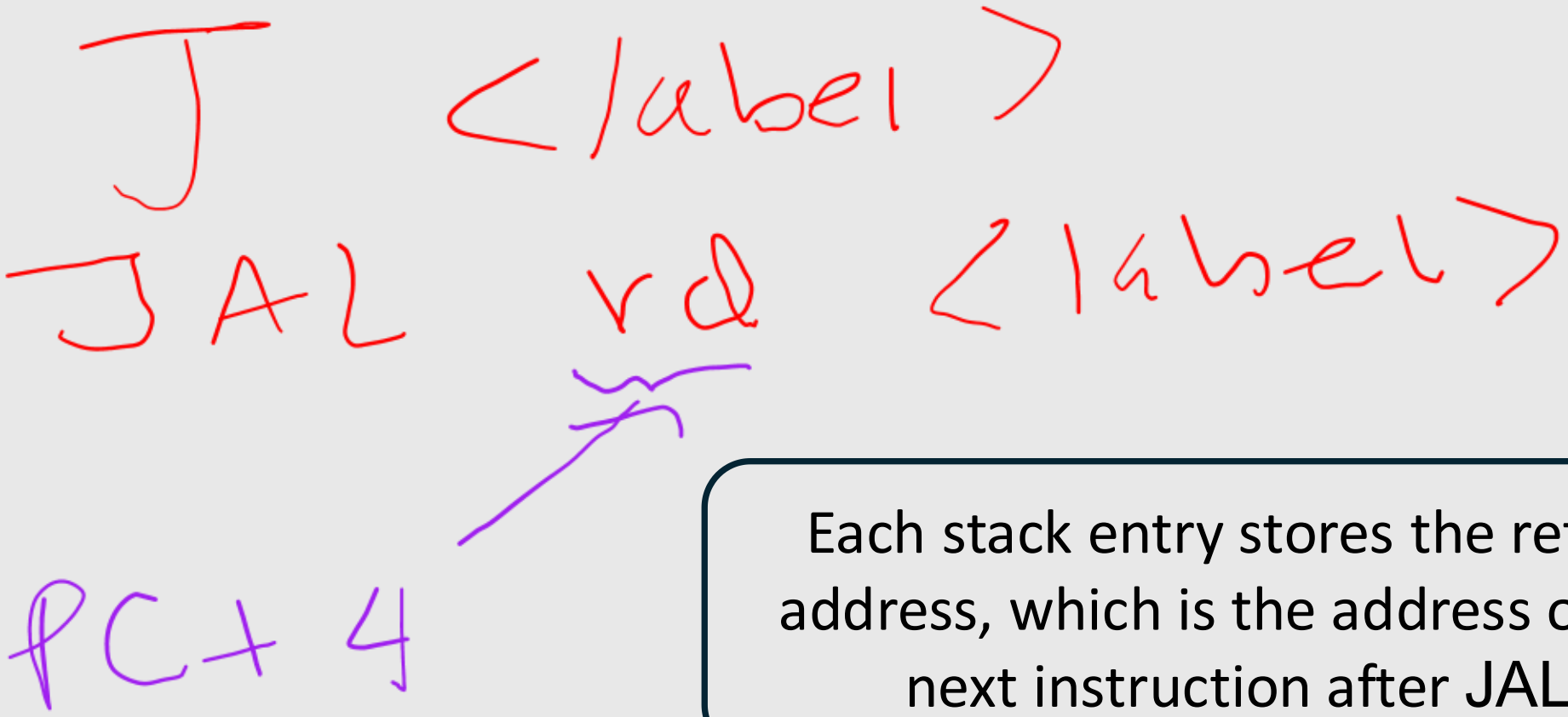
Old Practice Time!

Old Practice Time!

gcd(300,165), gcd(135,165),
gcd(135,30), gcd(105,30),
gcd(75,30), gcd(45,30),
gcd(15,30), gcd(15,15)

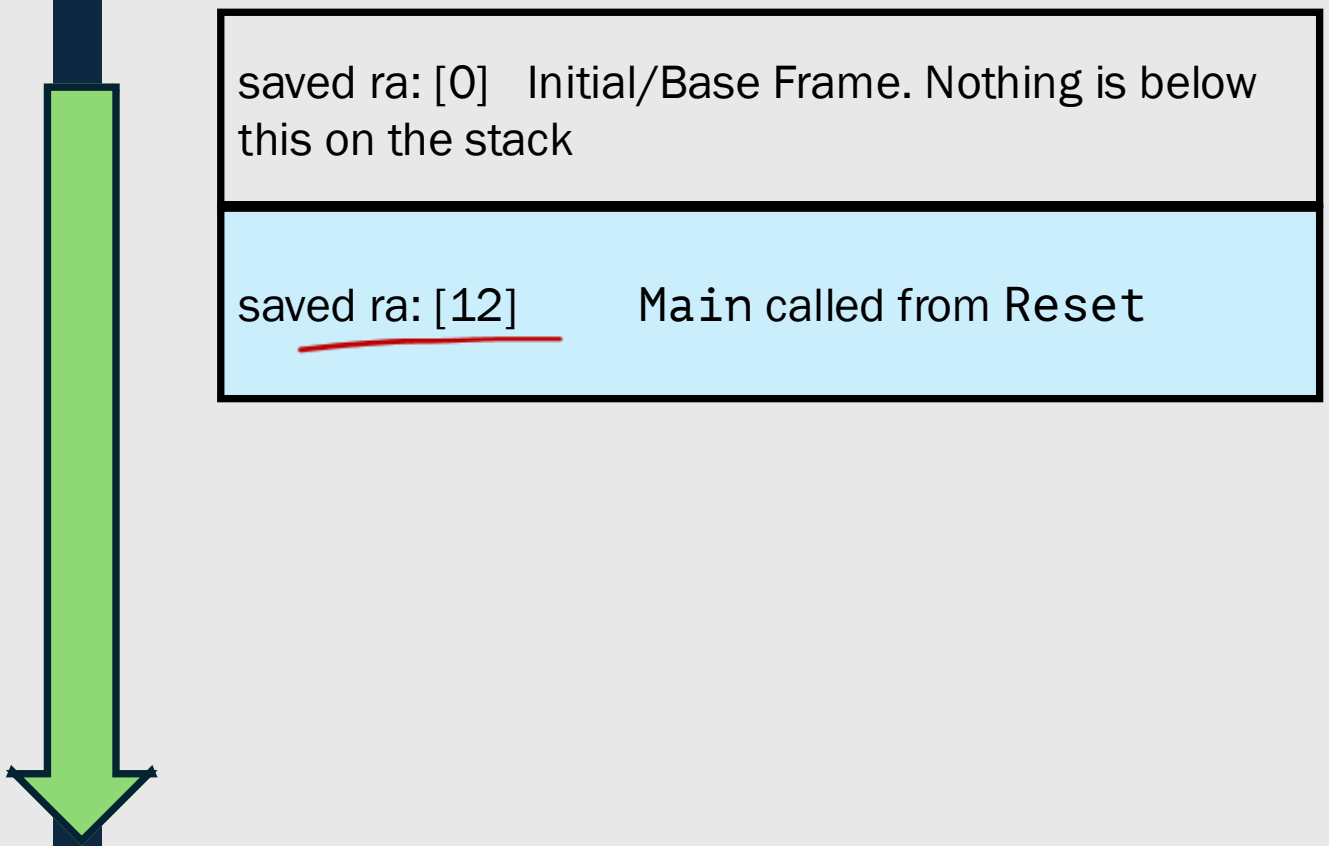
Stack Tracing. Following along w/ WS Code

saved ra: [0] Initial/Base Frame. Nothing is below this on the stack



Each stack entry stores the return address, which is the address of the next instruction after JAL

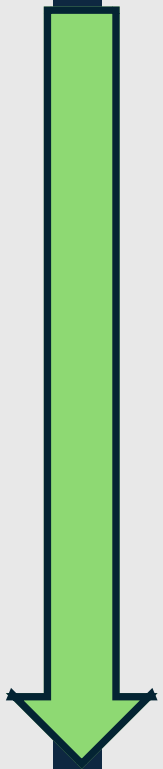
Stack Tracing. Following along w/ WS Code



addr 8: `jal ra, main`

PC+4

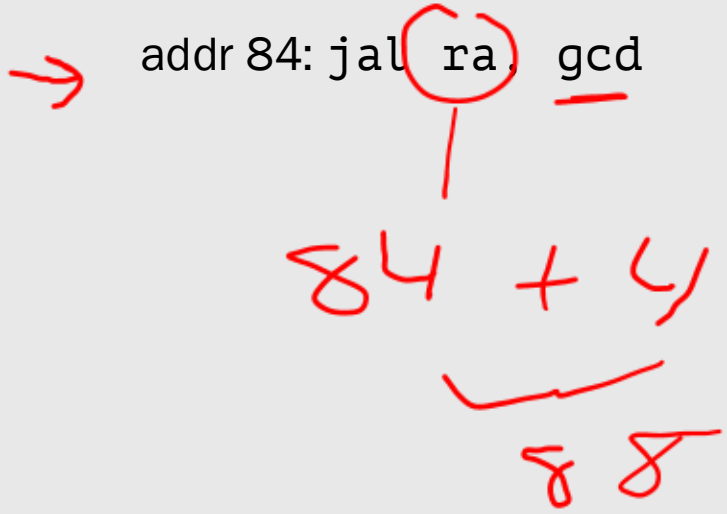
Stack Tracing. Following along w/ WS Code



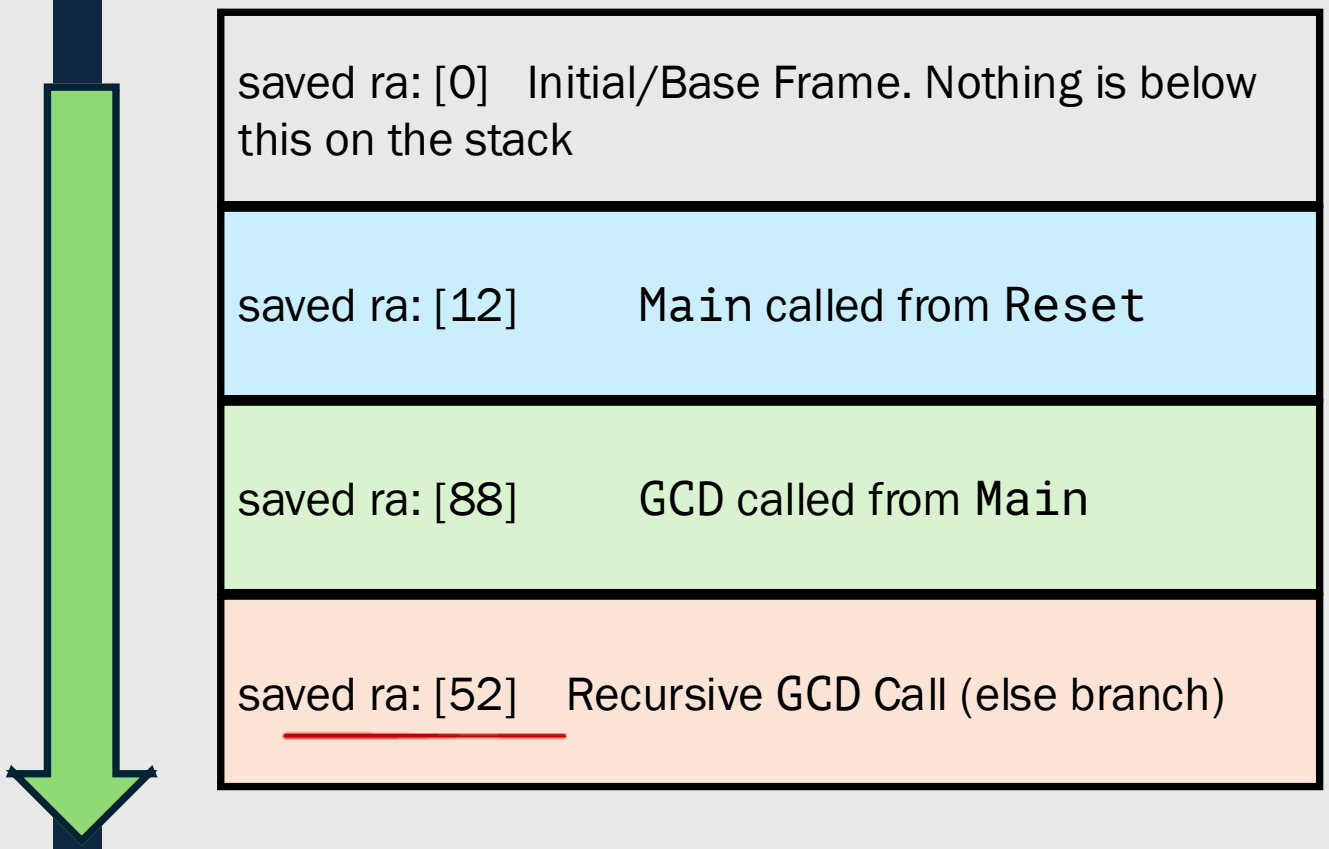
saved ra: [0] Initial/Base Frame. Nothing is below this on the stack
saved ra: [12] `Main` called from `Reset`
saved ra: [88] `GCD` called from `Main`

addr 8: `jal ra, main`

addr 84: `jal ra, gcd`



Stack Tracing. Following along w/ WS Code

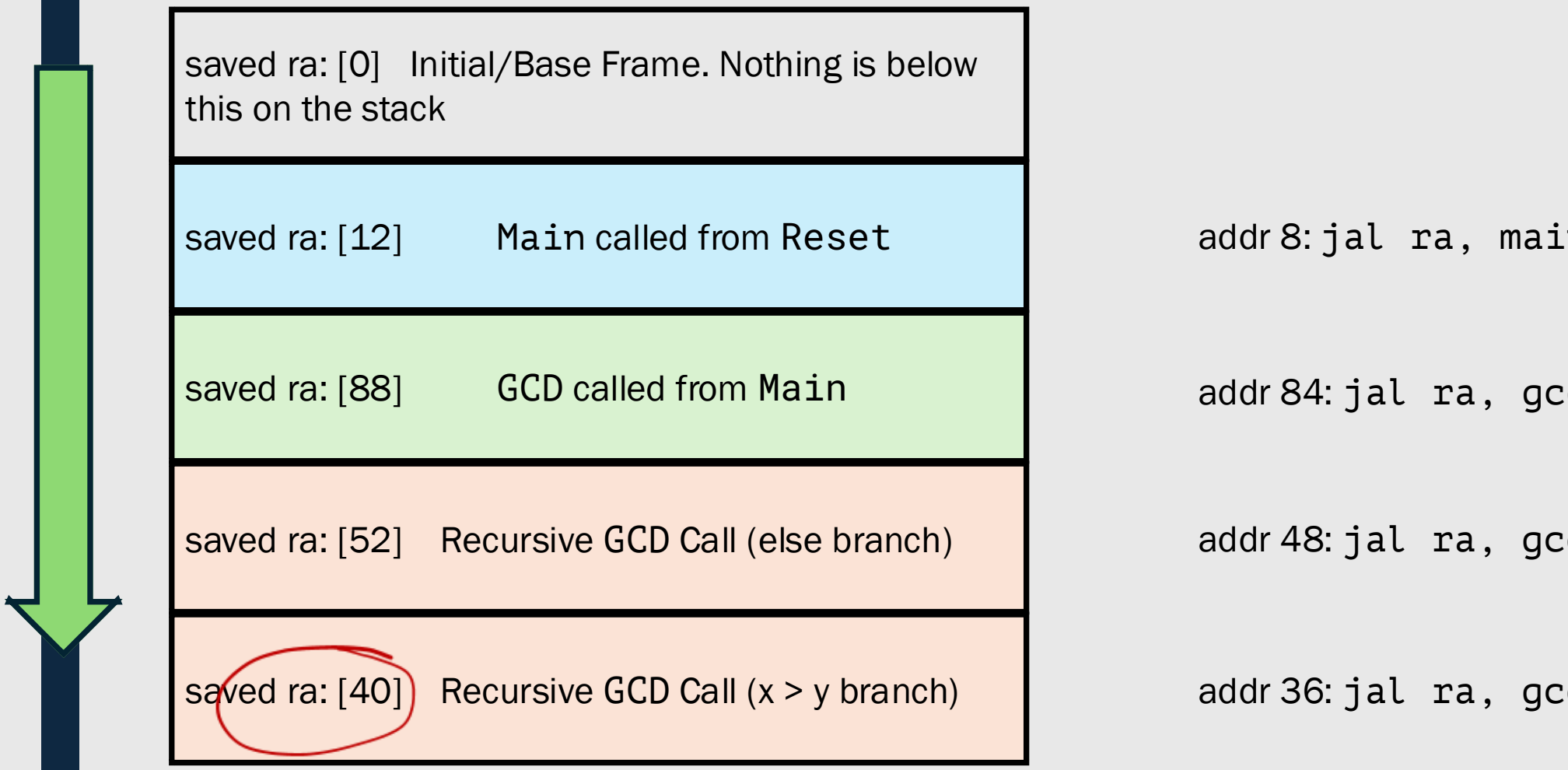


addr 8: `jal ra, main`

addr 84: `jal ra, gcd`

addr 48: `jal ra, gcd`

Stack Tracing. Following along w/ WS Code



addr 8: `jal ra, main`

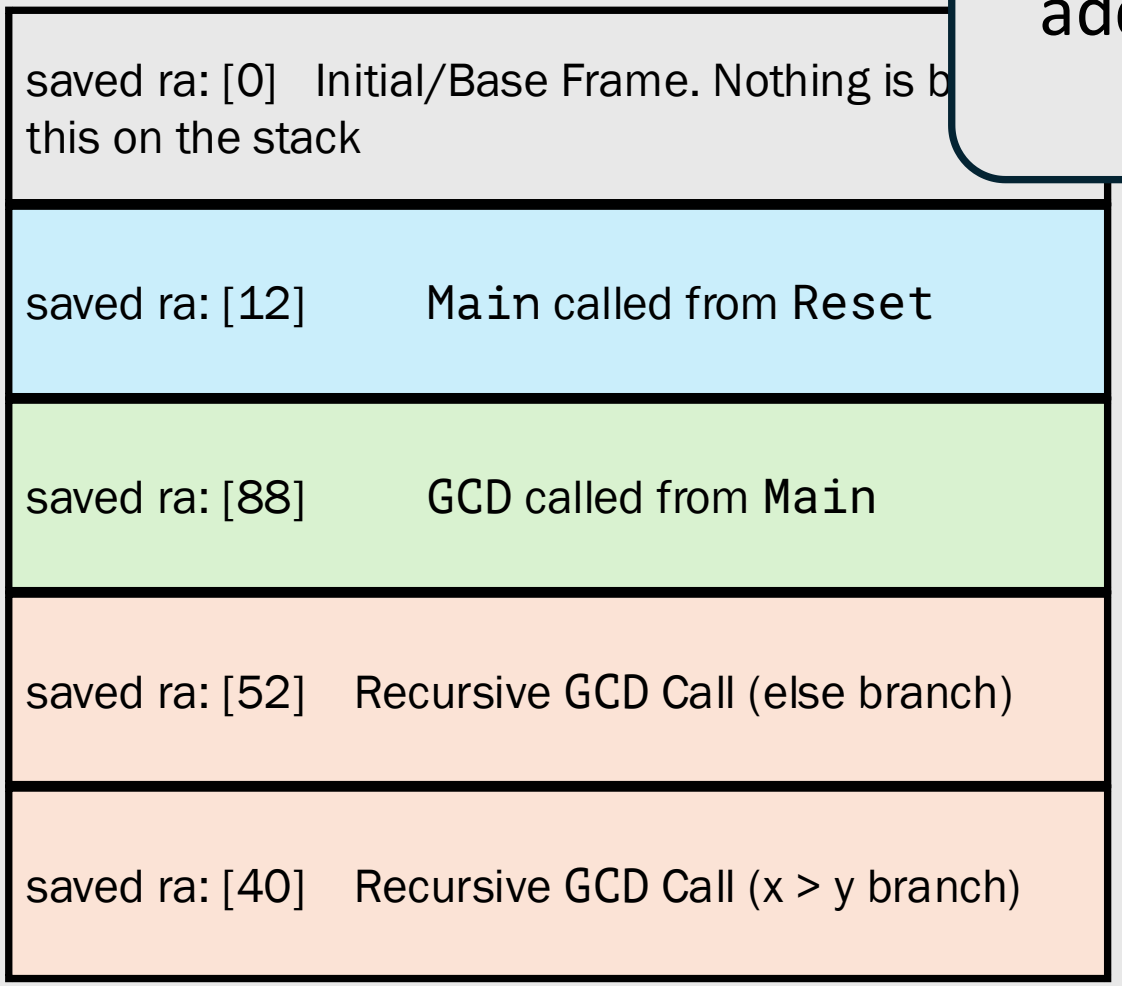
addr 84: `jal ra, gcd`

addr 48: `jal ra, gcd`

addr 36: `jal ra, gcd`

Stack Tracing. Following a

Each stack entry stores the return address, which is the address of the next instruction after JAL

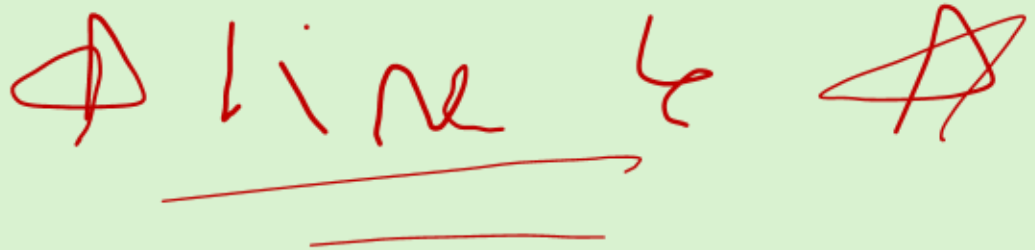


addr 8: `jal ra, main`

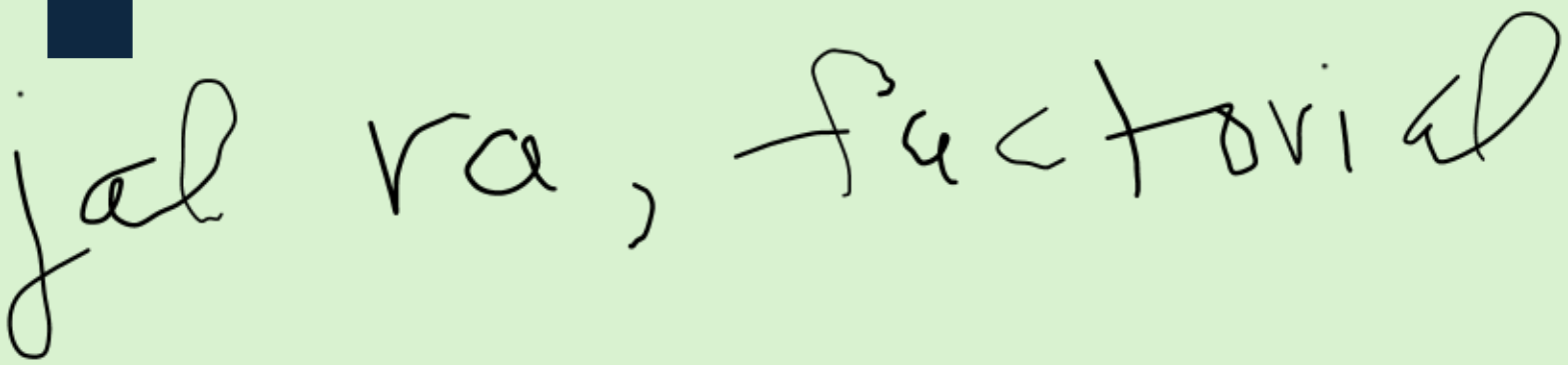
addr 84: `jal ra, gcd`

addr 48: `jal ra, gcd`

addr 36: `jal ra, gcd`



RISC-V BUG FIXING!



Bugs!

þeq a0, x0, recur
If n == 0, we should go to the **base case**, not the recursive case.
So it should branch to the base case.
Correct idea:
beq a0, x0, base
or restructure the code so recursion is skipped.

Line 11 error
j factorial
This performs a jump but **does not save the return address**.
Recursive calls require saving the return address so the function can return.
Correct instruction:
jal ra, factorial